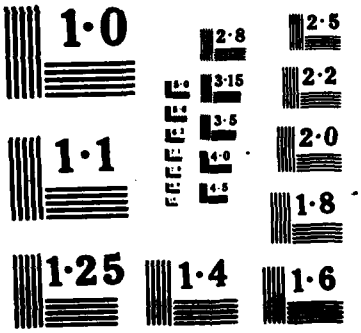
AD-A170 588 COMPILE-TIME PARTITIONING AND SCHEDULING OF PARALLEL PROGRAMS EXTENDED SUMMARY(U) STANFORD UNIV CA COMPUTER SYSTEMS LAB V SARKAR ET AL. 1986 MDA903-80-C-0432 1/1

UNCLASSIFIED F/G 9/2 NL


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6

Compile-time Partitioning and Scheduling of Parallel Programs

Vivek Sarkar and John Hennessy
Computer Systems Laboratory
Stanford University

Extended Summary

Contract # MDA903 80 C 0432

1. Introduction

One of the biggest challenges facing language designers and implementors is to develop languages that will be suitable for use on multiprocessors. A wide variety of multiprocessor architectures are currently being built and that variety is expected to increase. Very little software is available to exploit parallelism and speed-up the execution of individual programs. What little work has been done, has focused largely on special-purpose parallelism (e.g. vectorization) or very limited classes of architectures. To understand the issues involved in taking advantage of parallelism and to evaluate architectures, we need to find compilation techniques for fairly general-purpose languages; these techniques need to be adapted to a wide variety of architectures. Only then will it be possible to compare various languages, alternative architectures, and their interaction with different applications.

To compile and execute a program in a parallel fashion on a multiprocessor, there are three fundamental problems to be solved:

1) Identification of potential parallelism - discovering parallel computations in the program.
2) Partitioning the program into tasks - each task represents a serially-executed component that can execute in parallel with other tasks.
3) Scheduling the execution of tasks - the tasks must be scheduled for execution on multiple processors.

While the first step must be done at compile-time, the second and third can be done at compile-time, run-time or some combination thereof. The amount of effort required in each step depends on the initial language representation and on the characteristics of the target processor. Some languages, such as dataflow or single-assignment languages [2] make parallelism easily visible, while starting with Fortran requires the extraction of potential parallelism. Likewise, if the target machine is a dataflow architecture, then the task size is predefined to be a single instruction, and all scheduling is done by the hardware at run-time. If the machine is a multiprocessor, then partitioning and scheduling may be more appropriately done at compile-time.

1.1. Overview of our approach

Our compilation system concentrates on the latter two problems. We assume that the program is initially represented in a form that indicates all potential parallelism, namely a dataflow graph [6]. In our system, dataflow graphs are obtained by translation from a single-assignment language. However, they could also be derived from systems such as Parafrase [13], that automatically extract parallelism. We briefly discuss the advantages of single-assignment languages in the next section.

Most multiprocessors that have been built so far are either tightly coupled, shared-memory machines, or loosely coupled, message-oriented machines. Our approach is designed to be

tractable on this wide class of architectures. We review some of the important properties of multiprocessors in Section 1.3. None of these machines provide support for fine-grained parallelism. Instead, the relatively high overhead of scheduling a task and communicating its inputs and outputs requires a larger grain size for good performance. This means that partitioning schemes, whether compile-time or run-time, must rely on estimates of both the computational effort and the scheduling and communication overhead involved in a particular task.

The last phase in the compilation/execution process is task scheduling. Generally, run-time scheduling of tasks on processors is required for applications where task execution times are unpredictable. We have found that compile-time scheduling is justified for a significant class of applications with fairly predictable execution times. This approach is extremely attractive because it nearly eliminates all scheduling overhead at run-time. It uses information about the relative frequency of various execution paths in the program, and can produce excellent results, when the relative values of these frequencies are reasonably independent of input data. The Bulldog compiler [7] also uses similar frequency information, but coarse-grained parallelism on multiprocessors is substantially less sensitive to frequency variations than fine-grained parallelism in SIMD machines, like their VLIW architectures.

Before we examine the partition and assignment phases of our compiler, let's look at the language approach and some characteristics of multiprocessors.

1.2. Why single-assignment languages?

Single-assignment languages [2] get their name from the "single assignment rule", which states that a variable can only be assigned to once in a program. Consequently, an assignment is just a definition; a variable is defined to be an expression (which may contain other defined variables). This is in contrast to procedural languages where a variable serves as a memory cell, since it may be assigned to more than once. This fundamental difference makes single-assignment languages free from side-effects and applicative in nature. The applicative nature makes it relatively easy to decompose functions into subfunctions, and to compose functions into larger units. This is useful for graph expansion and partitioning.

An important consequence of their applicative nature is the parallelism present in single-assignment languages. All operations are implicitly parallel. Sequentiality among operations only arises due to data dependencies, when an operation's input is generated by a previous operation's output. This parallelism has been exploited at the instruction level in dataflow systems like the Manchester Prototype Dataflow Computer [11]. A single-assignment language allows one to concentrate on partitioning and scheduling, since parallelism is inherent in the language.

There are many ongoing research efforts in the area of single-assignment languages. Id

[3] combined the single-assignment principle with streams and iterations. VAL [1] excluded streams, but introduced types at compile-time. SISAL [15] is similar to VAL, but contains streams. SAL [5] has streams and iterations, but its types are restricted for run-time performance efficiency (e.g. no dynamic arrays, window buffer for streams). Despite differences in choice of features, all these languages have in common the fact that they are truly applicative. This is unlike some previous applicatively-oriented languages (e.g. LISP, PROLOG), that were forced to sacrifice freedom from side-effects for the sake of a practical implementation. Most criticisms (e.g. [8]) of single-assignment languages have in fact been criticisms of the practicality of dataflow machines. We, instead, implement these languages on coarse-grained multiprocessors.

Among applicative languages, single-assignment languages appear to be the most conducive to compile-time analysis. This is achieved at the expense of generality. Single-assignment languages differ from symbolic languages, such as LISP and PROLOG, in two fundamental ways (in addition, to being purely applicative):

1. Most single-assignment languages do not support functions as first-class objects, i.e. there are no functionals.

2. Single-assignment languages are usually strongly-typed with compile-time type checking.

These features allow for efficient compilation and implementation techniques similar to those used in conventional languages, such as Pascal and C. Thus, we expect single-assignment languages to be more appropriate for numeric computation, rather than symbolic processing that benefits from features in LISP and PROLOG.

1.3. Multiprocessors

Multiprocessors are general-purpose MIMD parallel machines containing several asynchronous processing elements. Due to the superior price-performance of VLSI-based uniprocessors, multiprocessors could easily become the mainframe and supercomputer machines of the future. We can classify multiprocessors as being "tightly coupled" or "loosely coupled". Tightly coupled multiprocessors (e.g. HEP [12], NYU Ultracomputer [9], the Sequent, Alliant and Encore machines) communicate through a shared main memory and, in some cases, a global inter-processor communication bus. Loosely coupled multiprocessors (e.g. the Cosmic Cube [16], the Transputer-based Computing Surface) communicate by exchanging messages through an interconnection network. In both cases, there is an overhead associated with inter-processor communication. This overhead has two performance factors:

1. Bandwidth - the amount of communication the system can support in unit time. Bandwidth limitations cause performance loss primarily through contention for the limited communication resources.

2. Latency - the delay between sending and receiving times of the communication, when the communication demand is less than the bandwidth.

Generally, tightly coupled multiprocessors have a smaller latency and a lower bandwidth, when compared to loosely coupled multiprocessors. The overhead of encoding and transmitting data

communication as messages and the possibility of multiple "hops" to reach a destination account for the larger latency in loosely coupled multiprocessors. The communication in tightly coupled multiprocessors is between a processor and shared memory or other processors, and is usually uniform for all processors.

The major cause of reduced communication bandwidth is simultaneous access to a single unit. In loosely coupled multiprocessors, this occurs during simultaneous communication to the same processor. In tightly coupled multiprocessors, this occurs during simultaneous access to the same memory unit or processor; this contention is potentially worse. If the shared main memory is also used to store unshared data (perhaps because the processor local memories are caches), then the communication bandwidth available for inter-processor communication is further reduced. The larger bandwidth available in loosely coupled multiprocessors allows them to support a larger number of processors, but with a greater latency, than in tightly coupled multiprocessors.

In partitioning programs and scheduling them for execution on a multiprocessor, the cost of communication becomes a major factor in deciding how much parallelism to exploit and how to organize parallel tasks on the processors. An appropriate balance between computation and communication is essential. Ignoring communication overhead could lead to an abysmally low performance; factors of 10 to 100 are common for communication latency, relative to the machine's cycle time.

1.4. Related Work

Omitted for brevity; discussion in this section would include mention of Kuck's PARAFRASE, Kennedy's vectorizing Fortran work, the Bulldog compiler, static allocation in the Hughes Data Flow Machine and early work on compiling SISAL for the Cray and HEP.

2. Multiprocessor scheduling with communication - the problem

The goal of compile-time processor assignment is to generate a "good" (close to optimal) schedule, taking parallel execution times and communication overhead into account. Existing theoretical models of multiprocessor scheduling do not consider communication costs, nonetheless the problem of finding the optimal schedule is NP-complete [14]. However, there exist efficient $O(n^2)$ scheduling algorithms that generate schedules with a performance bound of two [10], relative to the optimal schedule. Thus, the scheduling problem is not an impediment to achieving linear speedup in multiprocessors.

Unfortunately, these simple models are inadequate for our purpose because they ignore the overhead of inter-processor communication. Most schemes for processor assignment attempt to separately optimize parallelism, communication and load balancing. We believe that these parameters should be combined into a single objective function. Therefore, we have extended

the multiprocessor scheduling model to include communication costs. Interprocessor communication is assumed to occur along edges of the precedence graph (i.e. the dataflow graph), whenever the producer and consumer tasks of an edge are assigned to separate processors. The data along an edge is considered available only when its producer task has completed execution. The problem is to find a schedule with the smallest completion time.

Communication cost has two components:

1. *Processor overhead* - The processor time that must be spent for processing each input and output edge of a task.
2. *Delay overhead* - The time taken for the actual data transmission. Both producer and consumer processors are free to execute other tasks during this delay.

The delay overhead expresses the fraction of work done by hardware other than the processor (e.g. interconnection network switches, snoopy cache) during inter-processor communication.

Thus, the time to execute a task on a processor is the sum of its execution time and the processor time required to fetch and store non-local inputs and outputs. A task can only start after its predecessors have completed execution and the appropriate time dictated by the delay overhead has elapsed. Synchronization between tasks is modeled as communication; the delay overhead enforces waiting due to synchronization.

We ignore the effect of communication demand on communication overhead in this model. This is valid when the demand is less than the available bandwidth. We feel that multiprocessors should be designed with sufficient bandwidth to meet the maximum demand, just as uniprocessor memory systems are designed to support the peak memory access rate.

3. Partitioning and Scheduling

This section details the scheme we use for compile-time partitioning and processor assignment. Our approach is to create enough parallelism in the top-level function graph and to schedule the processors on that function graph so as to minimize computation time, while considering communication overhead. There are four basic steps in this process:

1. Traverse the graph and assign execution time costs to nodes and communication size costs to edges.
2. Expand the graph so that the main function contains sufficient parallelism to keep all processors busy.
3. Decide which edges in the expanded graph should always be internalized, i.e. not cross processor boundaries. This partitions the nodes into blocks so that all nodes in the same block must be assigned to the same processor.
4. Assign the nodes in the expanded, annotated graph to processors, while obeying the internalization constraints produced in the previous step.

Note that we rely heavily on compile-time estimates of communication and computation costs. These costs drive the expansion, internalization and assignment of parallel tasks. We have found

that our cost estimates yield good partitions for a range of programs and input data.

The 4 steps are described in the following subsections. Due to space limitations in this summary, we have omitted detailed descriptions of the algorithms and results. Instead we present brief descriptions of the basic methods used.

3.1. Cost assignment

This phase evaluates communication and computation costs in the dataflow graph. Communication costs are determined by examining the data type of an edge and assessing its size in an appropriate unit (e.g. bytes, words). Estimation of node execution times is more difficult and is undecidable in general. The unknown parameters are:

- The number of iterations in a While or Forall.
- The frequency distribution of the alternatives in a selection.
- Array size for nodes that operate on entire arrays.
- Recursion depth for recursive function calls.

Average node execution times are determined by using average values for these parameters. These average values can be estimated using simple rules of thumb, can be provided by the programmer through pragmas, or can be derived from profile information. Given these parameters, it is a straightforward task to compute the cost of a node from the cost of its components.

3.2. Graph expansion

Graph expansion is necessary to expose sufficient parallelism in the "main program" function, for compile-time scheduling. It only considers node execution times and ignores communication costs. The goal is to produce a graph with suitable parallelism for the given number of processors. The Edge Internalization phase may choose to ignore some of this parallelism because of excessive communication overhead. Costs allow this trade-off to be expressed quantitatively.

The graph expansion algorithm continues to expand nodes till no further expansion could possibly improve the parallel execution time of the graph. Function calls and compound nodes (e.g. If, While, Forall) are candidates for expansion. A Call node is replaced by a copy of the callee's function graph; this is similar to procedure integration in conventional languages. The recursion depth is a limit on the expanded call depth of a recursive function. A Forall node is expanded into sub-Forall nodes, as shown in Figure 3-1 for 4 processors. Each element A[i] can be computed in parallel; in the SISAL [15] example shown, A[i] is set to the square root of the integer i. The SISAL statement is translated to a Forall node in the dataflow graph. Our algorithm expands the Forall node and replaces it by the subgraph shown below it. The SplitRange node produces two arrays, LOW[1..4] and HIGH[1..4], which define 4 subranges for

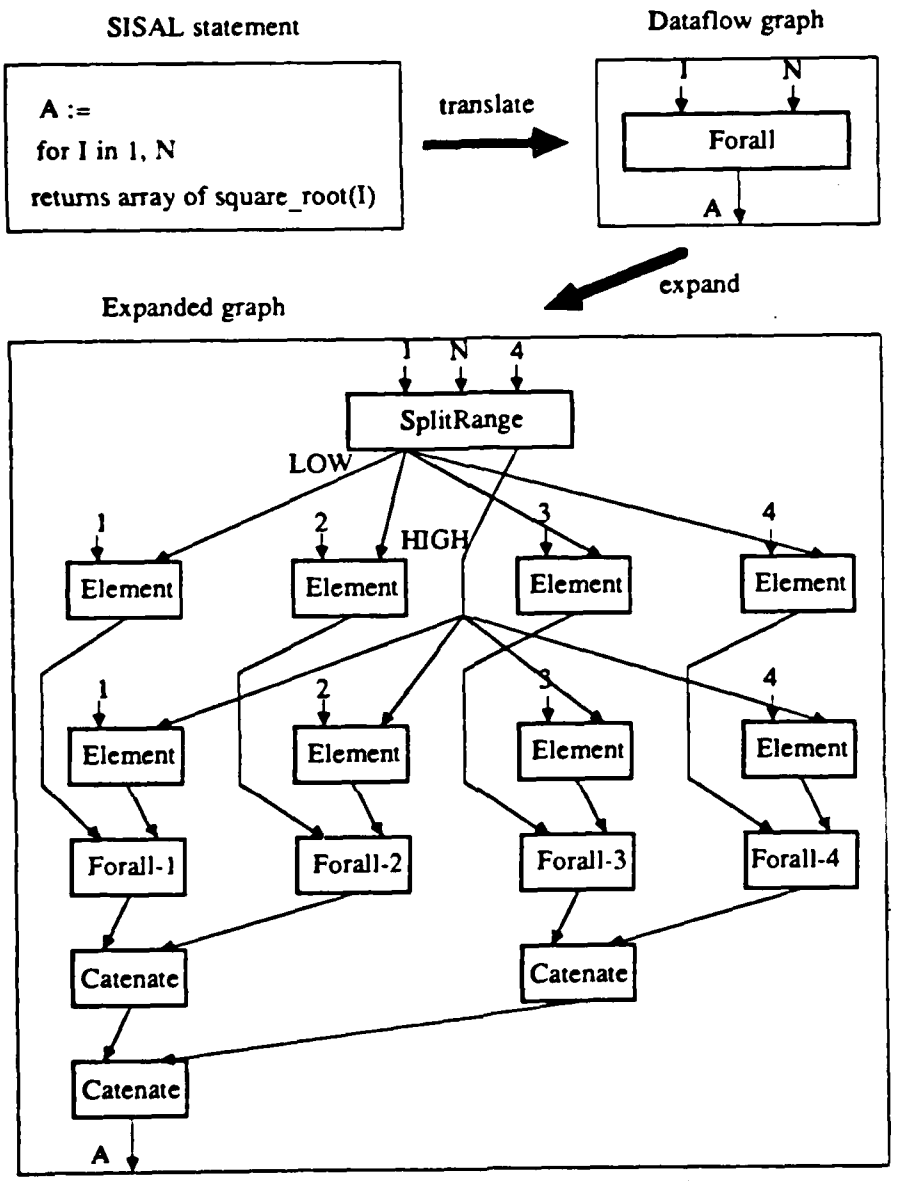


Figure 3-1: Expansion of a Forall node

the 4 sub-Forall nodes. The Catenate nodes are used to combine the 4 sub-arrays to yield array A. Note that the sub-Forall nodes are exact copies of the original Forall.

In graph expansion, we cannot afford to be thwarted by an If or While compound node at the outer level. Our algorithm expands an If into a sequence of its Condition, True and False graphs. It forces a pseudo-dependency between the True and False graphs so that they do not appear to

be in parallel. A While node is similarly expanded into its Initializer, Test, Body and Returns graphs.

The algorithm terminates when the graph has sufficient parallelism or when no node can be further expanded. A graph has sufficient parallelism if no node is a "bottleneck". Node N is a bottleneck if and only if the total cost of all nodes that can be executed in parallel with it is insufficient to keep P-1 processors busy, when N is executing, A bottleneck node is a candidate for expansion. The expansion algorithm proceeds by iteratively expanding the bottleneck node with the largest cost.

3.3. Edge Internalization

Edge Internalization is a pre-pass to Processor Assignment. Processor Assignment is performed by a single top-down pass of the graph, without backtracking. Though it reduces communication overhead locally, its lack of foresight in the absence of Edge Internalization could force some later edges to be external, no matter how large their communication overhead. Edge Internalization provides Processor Assignment with the foresight it needs, by letting it know *a priori* the critical edges that must be internalized.

The problem is to determine the set of internalized edges that yield the shortest critical path. Note that internalized edges can reduce the critical path length by reducing communication overhead, but can increase it by sequentializing parallel nodes. We have determined that the problem of finding the optimal set of internalized edges is NP-complete. We use a greedy algorithm that produces a critical path length within a factor of two of the optimal value. In each iteration, it internalizes the edge that yields the largest reduction in the critical path length, and terminates when no further reduction is possible.

3.4. Processor Assignment

We use a simple Priority List scheduling algorithm [10] for the final assignment of nodes to processors. It is adapted to our communication model by the following extensions:

1. It obeys the internalization constraints produced by the Edge Internalization phase.
2. It adds the processor overhead of communicating non-local inputs and outputs to a node's execution time.
3. It includes the delay overhead of communicating a node's non-local inputs, when determining the earliest time it can start execution.

Figure 3-2 shows the result of this phase on the expanded Forall graph in Figure 3-1. We've used simple values for node execution times (in ( )'s) and edge communication sizes (in [ ]'s), assuming N=100. The shaded areas show the assignment of nodes to the 4 processors. Edge Internalization forced all the Element nodes to be assigned to the same processor, because their communication overhead (=4) dominates their execution time (=1).

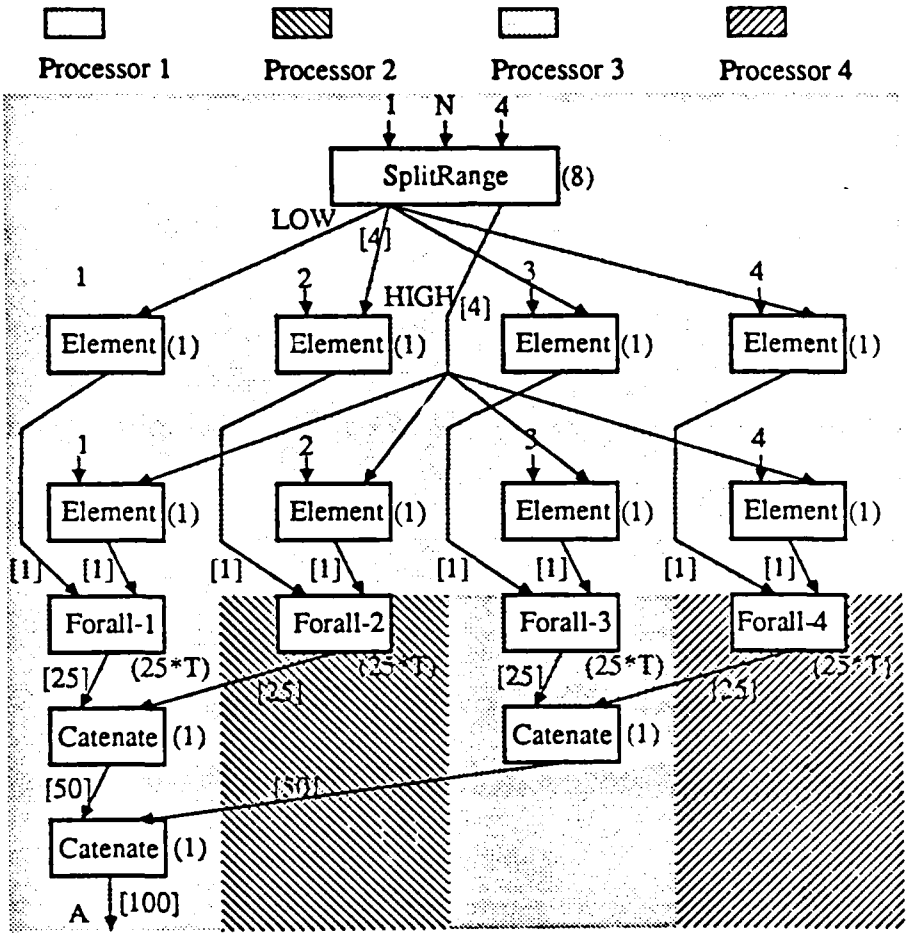


Figure 3-2: Costs and processor assignments of an expanded Forall

3.5. Code Generation

Once the partitioning into parallel tasks and assignment to processors has occurred, the code generation process is similar to that of a uniprocessor. Each node sequence is translated to sequential code for one of the processors. This translation must introduce synchronization primitives and communication code wherever an inter-processor edge appears. The code generator must preserve the order of nodes that have inter-processor edges. Arbitrary rearrangement could form a cycle in inter-processor synchronization and lead to deadlock during execution. Even if the reordering of such nodes avoided deadlock, it would be a different schedule from the one chosen by our algorithm, and may have a larger parallel execution time. However, the code generator is free to reorder and optimize instructions that do not cross an external synchronization or communication. There is expected to be a large scope for such conventional code optimizations, since the parallelization is performed at the outer level, and each node can have a lot of computation buried inside it.

4. Preliminary results

The 4 phases described in the previous section have been implemented to partition IF1 [17] dataflow graphs. A SISAL [15] to IF1 front-end allows us to process graphs produced by actual single-assignment programs. We have instrumented the Livermore IF1 interpreter to provide statistics for a multiprocessor simulation. The simulation uses processor assignments generated by our compile-time partitioning program. Execution time values are based on actual run-time frequencies and data sizes.

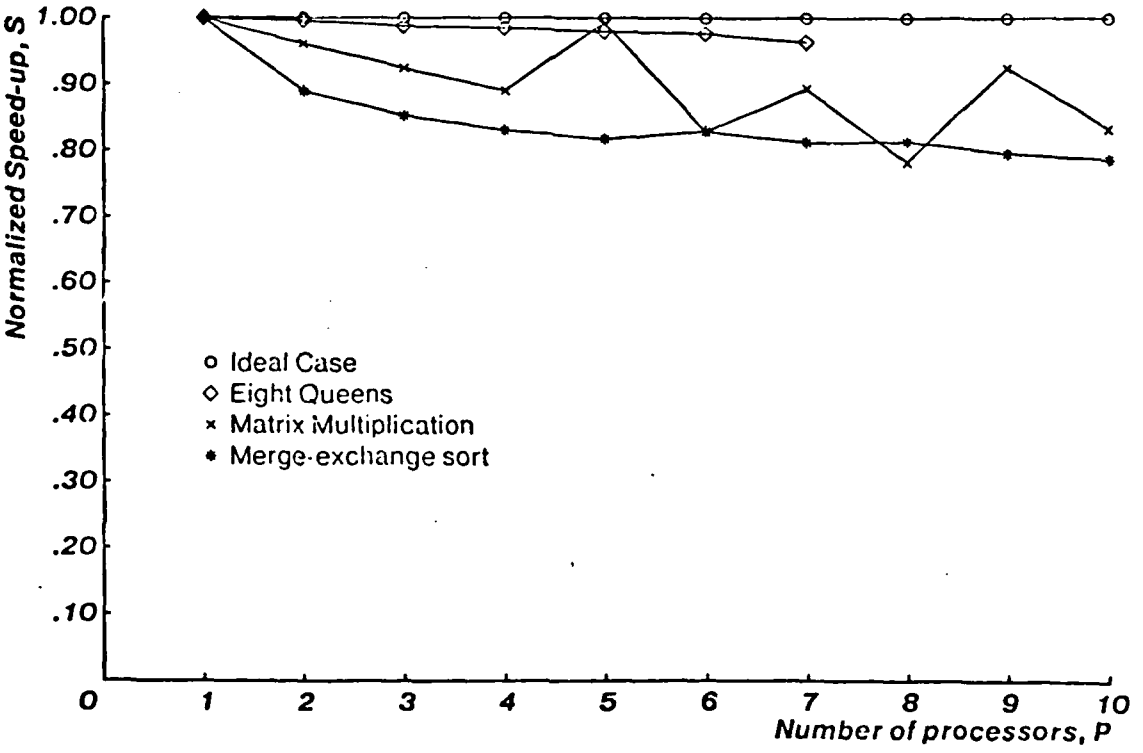


Figure 4-1: Normalized Speed-up vs. Number of Processors

Figure 4-1 shows some preliminary results obtained for the following programs:

1. *Matrix multiplication of two 25x25 integer matrices.* The parallelism was obtained by expanding the outer Forall, which has 25 iterations - one for each row in the first matrix.

2. Batcher's iterative merge-exchange sorting algorithm [4] on 100 integers. This is an excellent algorithm for parallel sorting. It consists of two nested While loops, each with log n iterations, and an inner Forall with n iterations. Graph expansion successively expanded the While bodies and finally the Forall, which contains the parallelism.

3. Eight Queens - a recursive program to generate all solutions to the 8 queens problem. A recursion depth value of 8 directed the graph expansion algorithm to expand the recursive call to 8 levels. The Forall at each level was then expanded.

The normalized speed-up (ratio of speed-up and number of processors) was found to be in the 75% to 100% range for these programs on 1-10 processors[1]. In general, the normalized speed-up drops as the number of processors increases because the proportion of sequential time becomes more significant. The seemingly erratic behavior of the curve for Matrix Multiplication occurs because of the relatively small number of iterations in its Forall. The parallel execution time of a Forall with N iterations on P processors is proportional to $\lceil P/N \rceil$; ignoring other computations, this leads to a normalized speed-up of $N/(P*\lceil P/N \rceil)$. This expression is a good approximation to the speed-up actually obtained for Matrix Multiplication, e.g. N=25 yields 0.89, 1.00, 0.83 for P = 4, 5, 6.

These programs were chosen as simple examples for an initial investigation. We are in the process of selecting bigger and more interesting examples from applications like Simulated Annealing, VLSI Routing and Computational Fluid Dynamics, and should have performance results for them in the near future. Also, this simulation data is presented for a single set of communication overhead parameters. We intend to perform all simulations for a range of communication overhead parameters representative of different multiprocessors, to study their effect on parallel execution time.

5. Conclusions

In conclusion we would like to make the following points:

- The partitioning and scheduling algorithms presented here are practical and have been implemented to process IF1 [17] graphs. We will use this implementation as a vehicle for further experimentation with different programs, input data and multiprocessor parameters.
- These techniques are machine-independent, as they do not assume a particular multiprocessor architecture. They are driven by a small table of parameters (e.g. number of processors, communication overhead factor, inter-processor distances), that describe the target multiprocessor.
- We have successfully combined communication overhead with parallel execution time in our model of multiprocessor scheduling with communication.
- We have an efficient method for determining execution time estimates, that goes beyond function boundaries and can also be used for recursive functions.
- We show how costs can be used in graph expansion to arrive at the right amount of parallelism.
- Optimal scheduling is NP-complete, but our Edge Internalization and Processor Assignment algorithms yield parallel execution times that are within a constant factor of the optimal value.

---

[1]For the Eight Queens program, we could not obtain data for all 10 processors in time for this summary, because of its large execution time on the IF1 interpreter.

References

1. Ackerman, W. B. & Dennis, J. B. VAL -- a value-oriented algorithmic language. Preliminary reference manual. MIT/LCS/TR-218, Laboratory for Computer Science, MIT, June, 1979.

2. Ackerman, W. B. "Data Flow Languages". *IEEE Computer 15*, 2 (Feb. 1982).

3. Arvind, Gostelow, K. P. & Plouffe, W. The (preliminary) Id report: an aynchronous programming language and computing machine. 114, Dept of Info & Computer Science, UC-Irvine, May, 1978.

4. Batcher, K. E. "Sorting networks and their applications". *1968 Spring Joint Computer Conf., AFIPS Proc. 32* (1968), 307-314.

5. Celoni, J. R. & Hennessy, J. L. SAL: A Single-Assignment Language for Parallel Algorithms. ClaSSiC-83-01, Center for Large Scale Scientific Computation, Stanford University, Sept., 1983.

6. Davis, A. L. & Keller, R. M. "Data Flow Program Graphs". *IEEE Computer 15*, 2 (Feb. 1982).

7. Fisher, J. A. *et al.* "Parallel Processing: A Smart Compiler and a Dumb Machine". *SIGPLAN Notices 19*, 6 (June 1984).

8. Gajski, D. D., Padua, D. K. & Kuck, D. J. "A Second Opinion on Data Flow Machines and Languages". *IEEE Computer 15*, 2 (Feb. 1982).

9. Gottlieb, A. *et al.* "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel COmputer". *IEEE Trans. Computers C-32*, 2 (Feb. 1983).

10. Graham, R. L. "Bounds on Multiprocessing Timing Anomalies". *SIAM J. Appl. Math. 17*, 2 (March 1969).

11. Gurd, J. R., Kirkham, C. C. & Watson, I. "The Manchester Prototype Dataflow Computer". *CACM 28*, 1 (Jan. 1985).

12. Kowalik, J. S. (Ed.). *Parallel MIMD Computation: HEP Supercomputer and Its Applications*. The MIT Press, 1985.

13. Kuck, D. J. *et al.* Dependence Graphs and Compiler Optimizations. Proc. 8th ACM Symp Principles Programming Languages, Jan., 1981, pp. 207-218.

14. Lenstra, J. K. & Rinnooy Kan, A. H. G. "Complexity of Scheduling under Precedence Constraints". *Operations Research 26*, 1 (Jan-Feb 1978).

15. McGraw, J. *et al.* SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2. M-146, LLNL, March, 1985.

16. Seitz, C. L. "The Cosmic Cube". *CACM 28*, 1 (Jan. 1985).

17. Skedzielewski, S. & Glauert, J. IF1 -- An Intermediate Form for Applicative Languages, Version 1.0. M-170, LLNL, July, 1985.

END

DTIC

9-86